Name: Sahib Bajwa
ID: 107553096

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 1 – Due Jan 24 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to LaTeX.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please submit this template of at least 9 pages (or Gradescope has issues with it).

Quicklinks: 1 2a 2b 2c 3 4a 4b 4c 4d

---

1. *What are the three components of a loop invariant proof? Write a 1–2-sentence description for each one.*

   The three components of a loop invariant are initialization, maintenance, and termination. The initialization component states that the loop invariant must be true at or before the first iteration of the loop. This is important because we want to make sure the invariant is true at the beginning of the loop before moving on. The maintenance component is used to show that if the invariant holds before the first iteration of the loop, it will hold at the start of consecutive iterations. Maintenance is important because we want tot make sure the invariant holds through the loop before it is exited. The final component is termination, which which is used to state the termination condition of the loop as well as use the invariant to say something useful about the state of the output. This component is important because it helps us show that the algorithm is correct and that the output is what we wanted the algorithm to do with the input.

Name: Sahib Bajwa
ID: 107553096

2. *Identify and state a useful loop invariant in the following algorithms. You* do not *need to prove anything about it.*

   (a)
```
FindMinElement(A) : //array A is not empty
    ret = A[length(A)]
    for i = 1 to length(A)-1 {
        if A[length(A)-i] < ret{
            ret = A[length(A)-i]
    }}
    return ret
```
   The variable ret is the minimum value from A[length(A)-i+1] to A[length(A)]. In other words, ret is always the smallest value in the array to the point we have searched/iterated to.

(b)
```
LinearSearch(A, v) : //array A is not empty and has no duplicates
    ret = -1 //index -1 implies the element haven't been found yet
    for i = 1 to length(A) {
        if A[i] == v{
            ret = i
    }}
    return ret
```

The sub array before A[i] does not contain the element we are searching for (v).

(c)
```
ProductArray(A) : //array A is not empty
    product = 1
    for i = 1 to length(A) {
        product = product * A[i]
    }
    return product
```

The absolute value of product * A[i] is $\geq |A[i]|$
$|product * A[i]| \geq |A[i]|$

Name: Sahib Bajwa
ID: 107553096

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 1 – Due Jan 24 11:55pm **Spring 2020, CU-Boulder**

3. *The algorihtm 2a is a standard find-min operation: it is supposed to return the element of minimum value in A. Use a loop invariant proof to show the algorithm 2a from the preceding question is correct.*

Here is a scaffold of the proof to get you started.

We will use the following as our loop invariant: The variable ret is the minimum value from A[length(A)-i+1] to A[length(A)]. In other words, ret is always the smallest value in the array to the point we have searched/iterated to.

Initialization: Before the first iteration of the loop, ret is = A[length(A)] and i = 1. A[length(a)-i+1] = A[length(A)], and since A[length(A)] is the only element and ret = A[length(A)], the loop invariant holds before the first iteration of the loop.

Maintenance: At the beginning of each iteration for the loop, the loop has check all values from A[length(A)-i+1] to A[length(A)] on whether they are less than ret or not. If any of the values were less than ret, the would have been set to ret. So the loop will have taken the min value from the sub array A[length(A)-i+1] to A[length(A)] and set it to ret. Thus, the loop invariant holds.

Termination: There is only one case in which the loop will exit. Case 1: i = length(A)-1. In this case, the loop has compared all values to ret, and if any were less than ret, ret was set to that value. Our loop invariant is that in the array A[length(A)-i+1] to A[length(A)], ret is the smallest value. Since the loop has checked all values in A, and ret is set to the smallest value in A, the loop invariant holds when the loop is exited.

Correctness: There is only one way for the algorithm to return, by reaching the end (or beginning I suppose) of the array. The loop only reaches the end of the array once it has compared ret to every value in the array and set ret to the smallest one. The algorithm then returns ret, the smallest value in the array A, correctly. The algorithm always returns the correct answer (the smallest value ret), and thus, the algorithm is correct.

Name: Sahib Bajwa

ID: 107553096

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 1 – Due Jan 24 11:55pm **Spring 2020, CU-Boulder**

4. *Let $A = [a_1, a_2, \ldots, a_n]$ be an array of numbers. Let's define a 'reverse' as a pair of distinct indices $i, j \in \{1, 2, \ldots, n\}$ such that $i < j$ but $a_i > a_j$; i.e., a_i and a_j are out of order.*

   For example - In the array A = [1, 3, 5, 2, 4, 6], (3, 2), (5, 2) and (5, 4) are the only reverses i.e. the total number of reverses is 3.

   (a) *Let A be an arbitrary array of length n. At most, how many reverses can A contain in terms of the array size n? Explain your answer with a short statement.*

   The most reverses an array A of length n can have is: [(n-1)*n] - [(n-1)*(n/2)]

   This answer comes from the fact that the most reverses possible will happen with arrays that are sorted from largest to smallest. When sorting like this, the maximum number of reverses turns out to be $\sum n - i$ from i=1 to n. Solving this summation for an equation gives [(n-1)*n] - [(n-1)*(n/2)].

(b) *We say that A is sorted if A has no reverses. Design a sorting algorithm that, on each pass through A, examines each pair of consecutive elements. If a consecutive pair forms a reverse, the algorithm swaps the elements (to fix the out of order pair). For instance, if your array A was [4,2,7,3,6,9,10], your first pass should swap 4 and 2, then compare (but not swap) 4 and 7, then swap 7 and 3, then swap 7 and 6, etc. Formulate pseudo-code for this algorithm, using nested for loops.* ***Hint:*** *After the first pass of the outer loop think about where the largest element would be. The second pass can then safely ignore the largest element because it's already in it's desired location. You should keep repeating the process for all elements not in their desired spot.*
Citation for help used: https://www.geeksforgeeks.org/bubble-sort/

```
SortSwapElements(A) :
    swap = 0
    for i = 1 to length(A) {
        for j = 1 to length(A)-i {
            if A[j] > A[j+1] {
                swap = A[j]
                A[j] = A[j+1]
                A[j+1] = swap
            }
        }
    }
    return A
```

(c) *Your algorithm has an inner loop and an outer loop. Provide the "useful" loop invariant (LI) for the inner loop. You don't need to show the complete LI proof.*

In the sub array from A[1] to A[j], A[j] is the largest value.

(d) *Assume that the inner loop works correctly. Using a loop-invariant proof for the outer loop, formally prove that your pseudo-code correctly sorts the given array. Be sure that your loop invariant and proof covers the initialization, maintenance, and termination conditions.*

We will use the following as our loop invariant: The sub array A[length(A)-i+1] to A[length(A)] is sorted and A[length(A)] is the largest element.

Initialization: Before the first iteration of the loop executes, i=1. The array A[length(A)-i+1] to A[length(A)] is one element, thus, the loop invariant holds.

Maintenance: At the beginning of each iteration of the loop, the largest value from the sub array A[1] to A[length(A)-i+1] has been set to A[length(A)-i+1]. This is done for all previous and future iteration of the loop. Thus, the loop invariant holds for all consecutive iterations of the loop.

Termination: The only case when the loop will exit is when i = length(a). at this point, the largest value of each loop has been set to its A[length(A)-i+1]. This means that when the loop exits, all values from A[1] to A[length(A)] have been sorted from smallest to largest with A[length(A)] being the largest. Thus, the loop invariant holds.

Correctness: The only way for the algorithm to exit is for i to equal A[length(A)]. The array will be sorted from smallest to largest values by this point, with A[length(A)] being the largest. The algorithm returns the sorted array A, correctly. The algorithm always returns the correct sorted array, thus the algorithm is correct.